Smartphone & Cross-platform Communication Toolkit User Manual

T4SM
ToolsForSmartMinds

Release 1.0.1
June 2011 Edition

Revision 1.0.1.60

Worldwide technical support and product information:
www.toolsforsmartminds.com
TOOLS for SMART MINDS Corporate headquarter
Via Padania, 16 Castel Mella 25030 Brescia (Italy)

Contents

Figure index

About this Manual

The Smartphone & Cross-platform Communication Toolkit User Manual describes the virtual instruments (VIs) used to communicate and pass data between LabVIEW and either a local or remote application. You should be familiar with the operation of LabVIEW, your computer and your computer operating system.

Conventions

The following conventions appear in this manual:

▸	The ▸ symbol leads you through nested menu items and dialog box options to a final action. The sequence **Tools** ▸ **Options** directs you to pull down the **Tools** menu, select **Options** item.
Bold	Bold text denotes items that you must select or click on the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Introduction

This chapter describes the installation procedure, installed components, and the main features of the Smartphone & Cross-platform Communication Toolkit.

Overview

The Smartphone & Cross-platform Communication Toolkit is an add-on package for communicating data trough applications. The toolkit contains a set of high level functions for sending your application data and advanced functions for customized tasks.

The following list describes the main features of the Smartphone & Cross-platform Communication Toolkit:

- Works over any TCP/IP connection
- Works over Local Area Networks as well as Internet connections.
- Implements the publisher – subscriber pattern (also known as Observer pattern)
- Authenticates subscribers through a API-KEY.
- Controls in background the state of every connection to identify loss of communication.
- Publishes GPS coordinates to manage mobile systems.
- Works with platform independent Data format and communicate with multiple platforms at the same time: third party vendors have implemented toolkit to develop on Android platform, Java, .NET and VB, Unix and iOs.

Because of the wide range of devices the Smartphone & Cross-platform Communication Toolkit works with, some portability issues remain. Consider the following issues when choosing your way to publish data:

- Some smart phones and tables uses CPU with low computing power so are not able to receive and process large streams of data.
- Smartphone & Cross-platform Communication Toolkit uses a platform independent data format and subscribers require a some computing power to decode data streams into their specific binary format.
- Smartphone & Cross-platform Communication Toolkit handles communication with subscribers as a set of peer to peer connections and every data you publish is transmitted individually to each subscriber. So you have to identify the right size of your data streams to avoid band saturation over your communication channel.
- Some data types are not supported on all platforms.

Installing the Smartphone & Cross-platform Communication Toolkit

Smartphone & Cross-platform Communication Toolkit is shipped as a VI Package Manager. You can download it from www.toolsforsmartminds.com.

Before installing Smartphone & Cross-platform Communication Toolkit you must install a copy of VI Package Manager on your machine. You can get a free copy of VIPM at this address:

http://www.jki.net/vipm/download

To install Smartphone & Cross-platform Communication Toolkit double click on

smartphone_&_crossplatform_communication_toolkit-1.x.x.xx.vip

and follow the installation wizard. Package contains LabVIEW Vis as well as documents in PDF format accessible from LabVIEW (**Help ▸TOOLS for SMART MINDS▸SCCT User Guide)** and libraries to create applications with Java, Android and .Net (in *c:\SCCT\cross-platform libs*). Visit http://www.toolsforsmartminds.com to get more details.

Top reasons to use Smartphone & cross platform Communication Toolkit

Adopting this toolkit you have the following advantages:

Don't re-invent the wheel: don't care about communication details over a TCP communication channel, toolkit does it for you.

Multiple platforms are supported: exchange your data with a protocol supported on a wide range of platforms and programming languages. In a near future

It's reliable: many applications have been created with this toolkit around the world.

Speed up your development activity: this toolkit simplifies the creation of distributed application and let you save a lot of your time.

Smartphone & cross-platform Communication Toolkit has been created by LabVIEW developers for LabVIEW developers: it includes some great features supported on LabVIEW platform only (see Publishing CustomData in this manual) so if you need to exchange data with other LabVIEW applications, take advantage of the power of this toolkit to deliver high quality code and reduce developing time.

Supported platforms

SCCT is composed by two main components:

- publisher library
- subscriber library

publisher library

this library let you create a full-featured publisher, which authenticates incoming subscribers, check connection status, sends data to all active publishers and passes their request to your application.

This library is available as a set of Vis for LabVIEW 2010 or later.

To get more details or download an evaluation copy of this library please visit:

http://www.ni.com/addontools

http://www.toolsforsmartminds.com

Subscriber library

This library let you create a subscriber which handles all communication details with a publisher so you don't have to. It receives data packages and present them to your application according to their data types.

This library is available for the following platforms and languages:

Name	***Operating System***	***Development Language***
SCCT Subcriber for LabVIEW[1]	Windows	LabVIEW 2010
SCCT Subscriber for VB[2]	Windows	Visual Basic, .NET,
SCCT Subscriber for Java	Java VM 5.0 or later	Net beans
SCCT Subscriber for Android	Android 2.1 or later	Java
SCCT Subscriber for Unix[3]	RedHat, Ubuntu, kernel build xxx	C
SCCT Subscriber for PowerPC[4]		C
SCCT Subscriber for iPhone/iPad[5]	iOs	Objective C
SCCT Subscriber for Phone7[6]	Phone7	.NET

To get more details or download your free copy of SCCT subscriber library, please visit:

http://www.ni.com/addons/SCCT

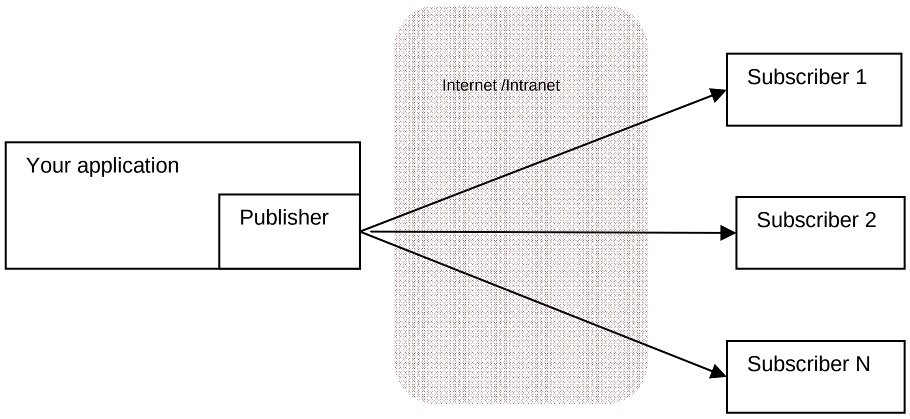
http://www.toolsforsmartminds.com

[1] This library is included in SCCT VIPM package and is not available separately from SCCT Publisher library.
[2] This library is distributed as OCX component and can be used by any language that support ActiveX technology.
[3] This library will be available from august 2011.
[4] This library will be available from august 2011.
[5] This library will be available from Q4 2011.
[6] This library will be available from Q4 2011.

Getting started with the Smartphone & Cross-platform Communication Toolkit

Communication concepts

Smartphone & Cross-platform Communication Toolkit implements the publisher-subscriber pattern. This well known pattern is also called Observer pattern. In this pattern you have one application (publisher) that receives the data you want to publish, and one to many applications (subscribers) which subscribe the service. To subscribe the service and receive fresh data from the publisher, they must use an API-KEY to be authenticated. The following figure represents the pattern:



When an application wants to receive data, asks the publisher to be inscribed among the active subscribers. Publisher will accept all incoming requests with the valid API-KEY.
When an application doesn't want to receive data anymore, simply inform the publisher. If the communication channel between publisher and subscriber fails, Publisher automatically removes the subscribers from its list of active listeners.

Using the Smartphone & Cross-platform Communication Toolkit

This Toolkit is composed of two main components: Publisher that creates the server side of your communication system and the subscriber that implements the client side. Publisher and Subscriber work together to pass data from one application which holds the data to many applications on different systems (MS-Windows OS family, Linux, Apple systems, mobile devices, etc.). Publisher uses a platform independent data format to transmit your data so that all subscribers can read them. Doing so you add a little overhead to a simple transmission that uses binary data format, but gain a great portability and opportunity to communicate with heterogeneous systems. To better understand the way this communication works, think of this example. A publishing company receives requests from different subscribers who want to receive a magazine. As long as they are subscribed, they receive the magazine. When they don't want to receive it anymore, simply cancel their subscription. Your application can implement more than one Publisher each of them works on different port of your machine. An application can contain publishers and subscribers together, working with different remote machines at the same time. Either objects work in background of your application with specific tasks that are created and destroyed automatically.

Publisher Class

Publisher is a Class with methods and properties detailed in the following tables:

Publisher Class properties		
Property name	*description*	*Read/write*
availableRequests	gets the count of received requests from subscribers	Read only
enableRequestList	Sets/gets the management of subscriber requests.	Read/Write
Port	Gets the actual port number	Read only
API-Key	Gets the actual API-Key	Read/Write
activeSubscribersCount	Gets the count of active connections	Read only
activeSubscribersAddresses	Gets the IP address and port of each active connection	Read only
Digital.publishOnlyOnChange	Sets/gets the property that defines how digital lines are published: if true, digital lines are transmitted only if their value is changed, otherwise they are transmitted every time regardless of their value.	Read/Write

Publisher Class methods	
method name	*description*
PublishData	Publishes analog and/or digital lines
startPublisher	Implements a publisher object
stopPublisher	Destroy a Publisher object
updateConfiguration	Transmit a new system configuration to all active connections
getRequest	Get the next request from one of the active connections
sendMessage	Transmit a message to a specific connection or to all active connections
notifyError	Transmit a message to a specific connection or to all active connections with a LabVIEW error code and source.
Read availableRequestCount	Returns the count of unprocessed requests.
Read activeSubscriberCount	Returns the count of active subscribers
Read activeSubscriberAddresses	Returns the IP address and TCP port of active subscribers
Read Port	Returns the current TCP port used by Publisher
enableRequestList	Enables the management of requests..
Read API-Key	Returns current API-Key
Write API-Key	Changes the current API-Key. Active connections are not affected by this change.
Read publishDigitalDataOnlyOnChange	Returns the property that defines how digital lines are published: if true, digital lines are transmitted only if their value is changed, otherwise they are transmitted every time regardless of their value.
Write publishDigitalDataOnlyOnChange	Sets the property that defines how digital lines are published: if true, digital lines are transmitted only if their value is changed, otherwise they are transmitted every time regardless of their value.

Creating a Publisher

To create a publisher in your LabVIEW application use the palette **SCCT ▸ Publisher**. The following examples show how to create a publisher in few click.
To create a publisher in your application you must choose two parameters that have to shared with subscribers:

- Publisher port is the TCP port that Publisher uses to manage all TCP connections.
- API-Key is the connection password that subscribers must communicate to publisher to be authenticated.

Take care to use one of the available port of your machine. Some ports are reserved for other common applications like port 21 to FTP, 80 to HTTP and so on. Moreover you have to check that the chosen port is open on your company firewall.

Publishing analog data

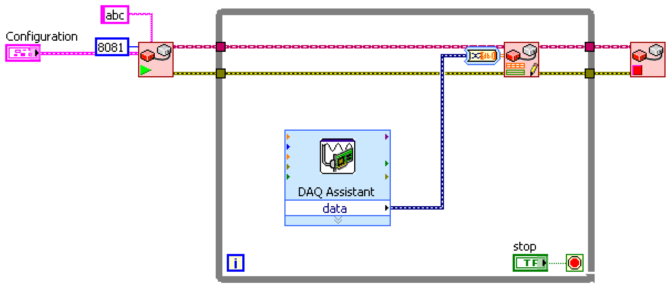


Figure 1 - Simple Publisher example with analog data.

In this example, a Publisher is created with **startPublisher.vi** that immediately creates all necessary data structures and tasks and takes care of all incoming connections. Please note that configuration cluster must be filled according to the analog signals you want to transmit. channelConfiguration must describe each channel of your data acquisition.
In the while loop your acquired data are published directly to the active subscribers. If no subscribers are connected, data are discarded.
When the loop terminates, **stopPublisher.vi** closes all active tasks and flushes FIFO with user requests.

*Notice: **startPublisher.vi** create some background tasks which handle the data transfer to and from active subscribers. These tasks are also responsible of checking if connections are lost in case your application is not publishing any information. All these tasks periodically check if your top level VI is running and stop automatically when your top level VI stops.*

Publishing digital data

In this example, a Publisher is created with **startPublisher.vi** that immediately create all necessary data structures and tasks and takes care of all incoming connections. Please note that configuration cluster must be filled according to the digital lines you want to transmit. digitalLines.Configuration must describes each line of your data acquisition. In the while loop your acquired data are published directly to the active subscribers. If no subscribers are connected, data are discarded. When the loop terminates, **stopPublisher.vi** closes all active tasks and flushes FIFO with user requests.

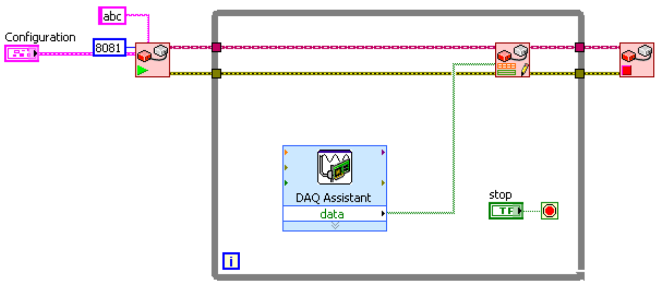


Figure 2 - Simple publisher example with digital data.

Subscriber Class

Subscriber is a Class with methods and properties detailed in the following tables:

Subscriber Class properties		
availableMessageCount	*Gets the number of received alerts from Publisher that application has to process*	*Read only*
availableAnalogData	Gets the number of analog data packets received from Publisher that application has to process	Read only
availableConfiguration	Gets the number of configuration clusters received from publisher that application has to process. Remember that when you establish a connection, you receive immediately a configuration cluster.	Read only
availableDigitalData	Gets the number of digital data packets received from Publisher that application has to process	Read only
connected	Returns TRUE if connection is active, FALSE if connection is lost	Read only
connectionStatus	Gets a numeric code related to connection status	Read only

Subscriber Class methods	
method name	*description*
openConnection	Creates a subscriber and open a connection with a running Publisher
closeConnection	Destroys a subscriber and close connection, if active
Read Message	Reads next available alert received from Publisher
Read analogData	Reads next available analog data packet received from Publisher. To activate data transmission, you must use **transferStatus.vi**
Read digitalData	Reads next available digital data packet received from Publisher. To activate data transmission, you must use **transferStatus.vi**
Read configuration	Returns the first unprocessed configuration cluster received from Publisher. After a connection is established, Subscriber receives a remote system configuration with a description of analog channels and digital lines, location and other system information.
Read customData	Returns the first unprocessed custom data cluster received from Publisher. Your application must be able to match the custom data received (a variant value) with the code specified by the application that sends the item.
discardData	Throws all received alerts, analog and digital data and configuration clusters away
transferStatus	Set the status of transmission. If you connect TRUE, publisher start sending analog data and digital lines as soon as they are available on server side. If you connect FALSE, Publisher stop data transmission. Alerts from publisher cannot be stopped.
Send Request	Sends immediately a request to the Publisher.
Read availableAnalogDataCount	Returns the count of unprocessed analog data packets.
Read availableDigitalDataCount	Returns the count of unprocessed digital data packets.
Read availableConfigurationCount	Returns the count of unprocessed configuration cluster packets
Read availableMessageCount	Returns the count of unprocessed messages
Read availableCustomDataCount	Returns the count of unprocessed custom data cluster packets
Read connected	Return a boolean value with the connection status. If TRUE then connection is active otherwise is FALSE.
Read connectionStatus	Returns a string that indicates the connections status or the failure reason in case openConnection doesn't succeed to establish a valid connection.

To implement the subscriber in your application and receive data from a source, you must know three parameters:

- Data source address, that is usually the IP address of the machine where Publisher is running on.
- Data source port, that is the TCP port of the Publisher.
- API-Key is the key necessary to be authenticated by publisher. If a subscriber uses a wrong API-Key connection is refused by publisher.

When your application succeeds to connect, publisher sends immediately a configuration of remote system i.e analog channel descriptions, unit of measure and range of all signals, digital line descriptions and system location GPS coordinates.

Creating a Subscriber

In the following example you create a simple subcriber with **openConnection.vi** that need three parameters: Publisher address (default value is localhost), Publisher port (default value in 8081) and API-Key.

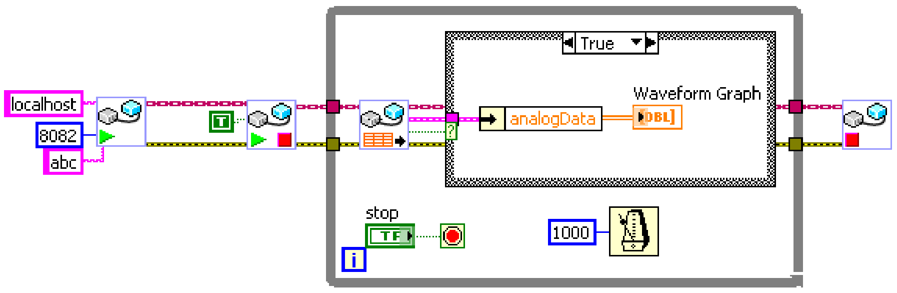
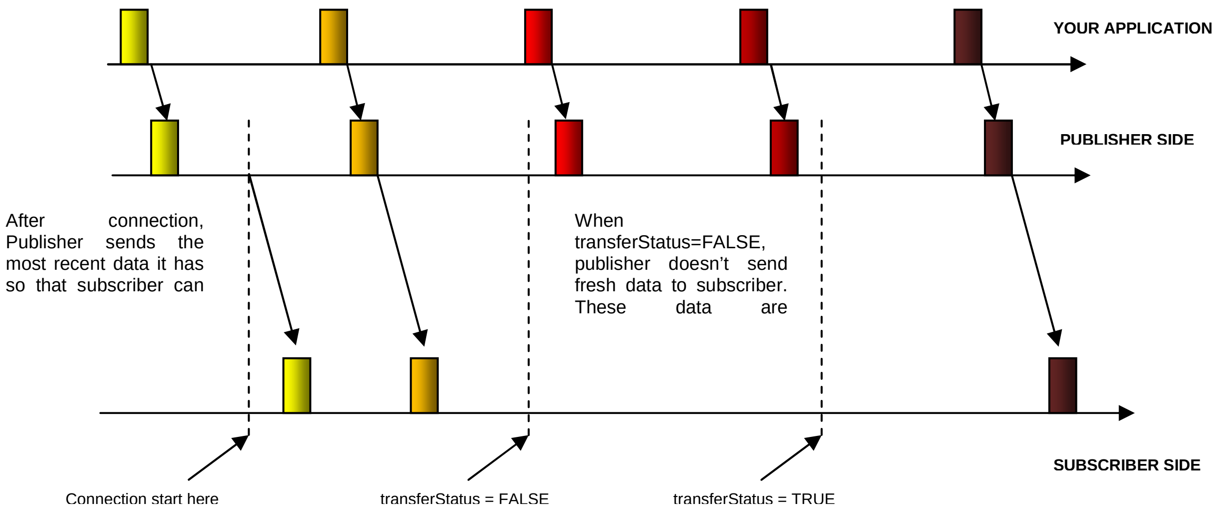


Figure 3 - Simple Subscriber Example that reads analog data only.

Publisher doesn't buffer data if transmission is stopped. The following diagram illustrates how analog and digital data are managed in situation.



YOUR APPLICATION
PUBLISHER SIDE
After connection, Publisher sends the most recent data it has so that subscriber can
When transferStatus=FALSE, publisher doesn't send fresh data to subscriber. These data are
SUBSCRIBER SIDE
Connection start here
transferStatus = FALSE
transferStatus = TRUE

Figure 4 – transferStatus.vi determines the flow of packets between publisher and subscriber.

Publisher doesn't start data transmission automatically. After connection, your subscriber must tell to the Publisher to start sending data. If your subscriber doesn't need fresh data, use **transferStatus.vi** with a FALSE constant to tell to publisher to stop sending data. To re-start data transmission use **transferStatus.vi** with TRUE constant.

To get analog data use **Read analogData.vi** that returns a packet of data. Background tasks takes care of all received packets and enqueues them in a FIFO so you can process all packets without data loss. To know the number of available packet use the property node that returns the data packets count. The following figure illustrate the case of reading digital lines only. Notice that at the end of the while loop you always have to close communication with publisher. When communication is closed, if you want to open again the communication use **openCommunication.vi.**

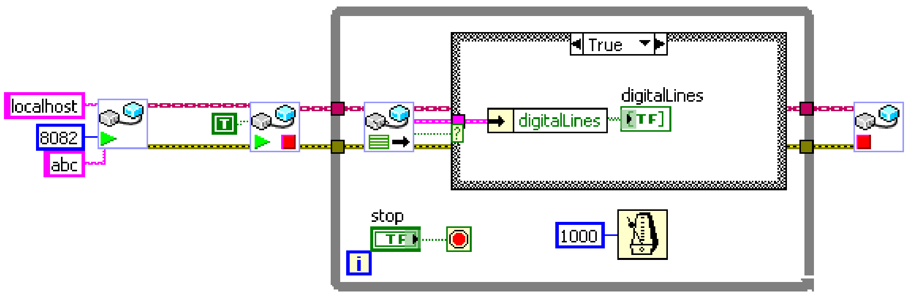


Figure 5 - Simple subscriber example that reads digital data only.

Publisher automatically sends a communication cluster that describes the remote system. Use this cluster to properly format your graph and chart setting x and y scales, as shown in the following figure.

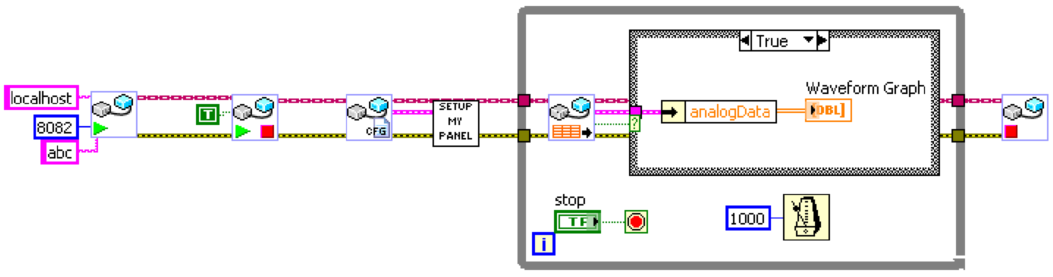


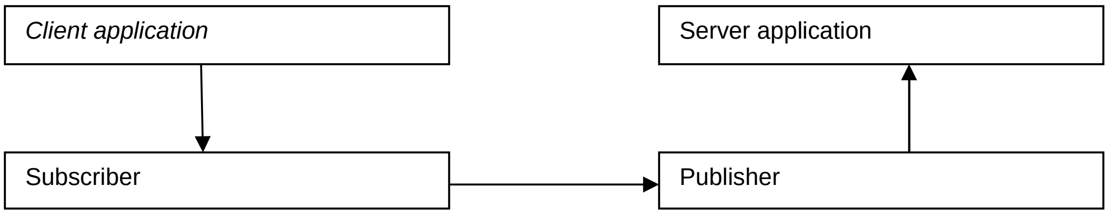
Figure 6 - Simple Subscriber that reads configuration cluster before reading analog data.

*Notice: **openCommunication.vi** create a background task which handles the data transfer to and from the publisher. This task is also responsible of checking if connection is lost in case publisher is not sending any information. This task periodically checks if your top level VI is running and stops automatically when your top level VI stops.*

Optionally you can add an application's description to the openConnection.vi. Server side application will use the description to identify your application. This is useful when the server has to properly identify subscriber's name, but they change IP address or port at every new connection, typically when DHCP is used to assign IP addresses. If description is not specified, server will identify the connection with the string "xxx.xxx.xxx.xxx:yyyy", xxx.xxx.xxx.xxx is subscriber application IP address and yyyy is local port used to connect to server.

Sending Requests

Subscribers can send textual requests to the Publisher with **sendRequest.vi.** In this context user means your application and a request is a string message that Publisher receives and passes directly to its main application. The following figure illustrates the path of user requests.



Requests are sent immediately to publisher and server processes them in the same order they are received. If string is empty, message is not posted. You can specify an additional numeric code if your application uses numeric codes to identify request sets.

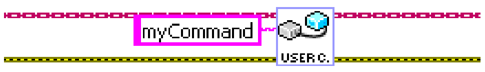
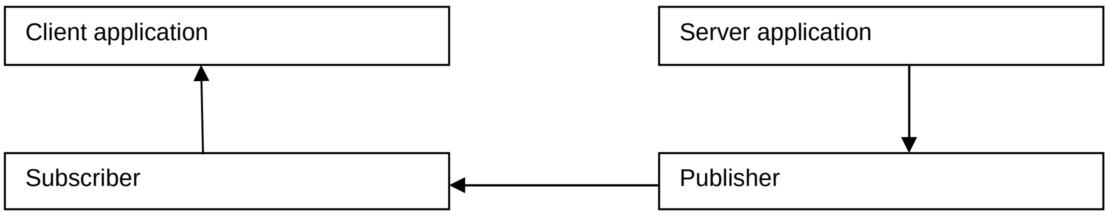


Figure 7 - sendRequest.vi example.

Receiving Messages

Subscribers can receive textual messages from the Publisher with **Read message.vi.** a message is a cluster composed by a timestamp, a string and a numeric code. The following figure illustrates the path of alerts.



Messages are received and enqueued in a dedicated FIFO. Your application have to process incoming messages and take care of FIFO size. You can use Read Message.vi to extract from the message FIFO the oldest received message.

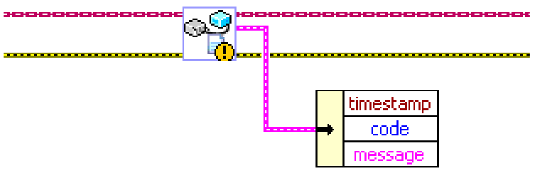


Figure 8 - Reading a message from Publisher.

activeSubscriberAddresses property node returns a 2D string array: every row contains IP address, port and description of an active subscriber. Rows are ordered by connection start time so first row regards the oldest active connection and last row regards the most recent connection.

How to check the state of your connection

You can monitor connection state with **connected** property node which returns TRUE if your connection is still alive. A connection is alive also if Publisher is not sending data to your subscriber. Publisher and subscriber exchange acknowledge packets to verify if connection is still active so you don't have to.

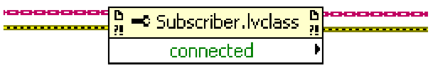


Figure 9 - Using properties nodes to check connection status.

Closing communication

When connection is no more necessary use closeCommunication.vi to close the open connection. This VI destroys all unprocessed data and closes background tasks. After this VI, subscriber object cannot be used and a new instance must be created with openCommunication.vi.

Creating and managing alerts and user requests

Overview

When an active communication is established between Publisher and subscriber, they can exchange some messages with a specific format: messages from your application to subscriber(s) are called Alerts. Every alert is composed by a numeric code and a message string. Messages from subscribers to your application are called user requests. Every user request contains a numeric code, a timestamp, a connection identifier, an event code and an optional data string. Publisher uses a FIFO to enqueue all incoming user requests in the order they are received. Your application can identify which subscriber is sending the request by its ipAddress:port identifier.

Reading available Requests

Subscribers send their request to your application and Publisher keeps them in a FIFO together with some messages it sends to inform your application about the connection status and the communication between publisher and subscribers. Use **availableRequests** property node to retrieve the number of received request that your application has to process. When a request is processed with **getRequest.vi**, **availableRequests** is decremented by 1.

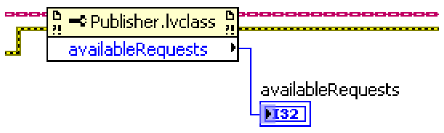


Figure 10 - Using available request property node to check is subscriber's requests are present.

Sending message to a specific subscriber

Your application can communicate with active subscribers with custom messages. To send a message use **sendMessage.vi**, as shown in the following figure:

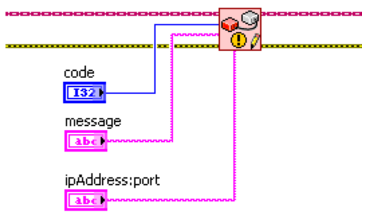


Figure 11 - Sending message to a specific subscriber with sendMessage.vi.

Use **message** string to add additional information to your message. To send a message to every active subscriber **ipAddress:port** must be empty string, instead, if you want to send a message to a specific subscriber, connect **ipAddress:port** to the specific address and port of its connection. Message string cannot be empty. If try to send an empty string message, error is generated with error code 5002.

Managing User Requests

User request (i.e. message from subscriber) is inserted into a FIFO by Publisher so your application can process all requests in the order they are received from Publisher. The following figure illustrates the right way to manage the requests:

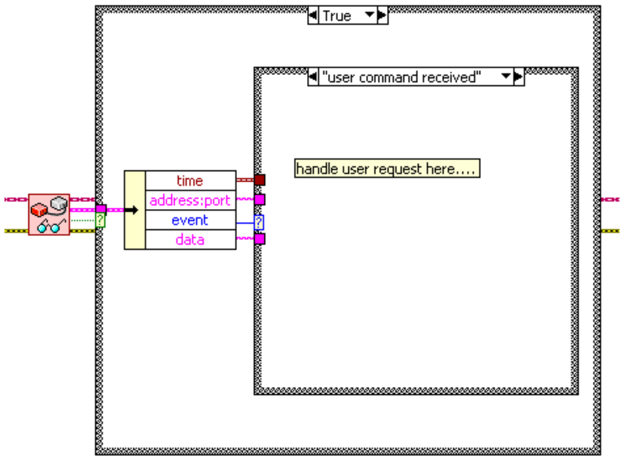


Figure 12 - Managing received request example.

First, you have to check that **found** indicator is TRUE, if **found** is FALSE, no request is available. All requests are classified with **event** field. User requests are returned with “user command received” value, as shown above. Every request has its **time** field, **address:port** field that identifies the subscriber that sent the request and an optional **data** field.

Notifying an error

When a LabVIEW error arises and must be notified to one or more subscribers, use notifyError.vi which composes a message with error code and error description from LabVIEW error cluster. The following example shows how to use it. Please note that you can specify a ipAddress:port reference to send error only to a specific listener. When error cluster contains no error, no message is sent.

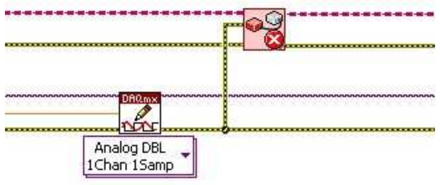


Figure 13 - Notifying LabVIEW error to all subscribers with notifyError.vi.

Advanced Options

Controlling active connections

Publisher Vis handle all incoming connections and close inactive connections so you just focus on your main application and forget all issues related to data transmission. In some cases you want to know the number of active connections and the address of subscribers. Usually you can map all incoming connections using getRequest.vi and filtering events such "connection successful" and "connection closed" and "connection timeout", which help you to map all active and closed connections.

Reading active connection count

To know the count of active connections at a specific time, use **Read activeSubscribersCount.vi** that returns the number of active connections.

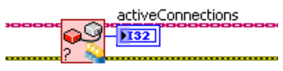


Figure 14 - Use Read activeSubscribersCount.vi to check the count of active connections.

Reading active connection addresses

To know the addresses of active subscribers, use **Read activeSubscribersAddresses.vi that** returns a string array: every string is a subscriber's address in the form ipAddress:port. the array is ordered by connection time so the first element of the array is related to the active subscribers that connected first. Closed connections don't appear in the array.

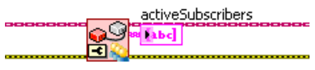


Figure 15 - Use Read activeSubscribersAddresses to get info about active subscribers.

You can use **activeSubscriberCount** property node to get active connection count and **activeSubscriberAddress** property node to get active connection addresses, as shown in the following figure:

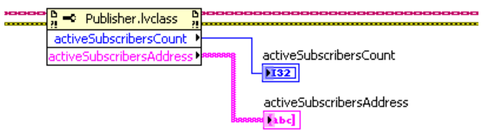


Figure 16 - Use property nodes to get information about active subscribers.

Publishing digital lines on change only

In many cases digital lines don't change very quickly so you can publish them only when they change. in this way you can reduce the consumed band on your communication channel. To change the way publisher works with your digital lines, use the **Digital.publishOnlyOnChange** property node. If you want to publish digital lines also if they aren't changed, set FALSE value to this property node. If you want to publish digital lines only when their value is different from the previous one you published, set this property value to TRUE. By default, a Publisher works with **Digital.publishOnlyOnChange** equal to FALSE:

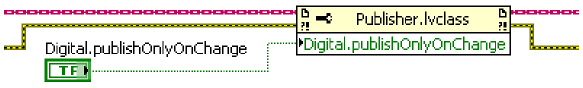


Figure 17 – Changing the way digital data are updated to all subscribers with Digital.publishOnlyOnchange property node.

If your application doesn’t need to handle user request you can disable userRequest FIFO to avoid waste of memory. To disable or enable FIFO, use **enableRequestList** property node, as shown in the next figure.

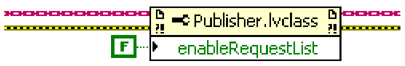


Figure 18 – Enabling the request FIFO with enableRequestList property node.

By default, every Publisher starts with FIFO enabled, so remember to consume elements in this FIFO with **getRequest.vi** or disable the FIFO with **enableRequestList** property node.
Many properties can be set either by specific Vis or by properties nodes. We encourage you to use properties node whenever it is possible.

Changing API-Key at run-time

Publishers are created with an API-Key they use to authenticate every incoming connection. You can modify the API-Key at run-time so that new subscribers have to use the new API-Key to be accepted. Existing connections are not affected by API-Key changes. To update the API-Key use **Write API-Key.vi** as shown in the following figure:

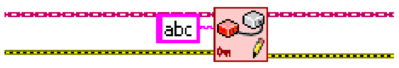


Figure 19 – Changing API-Key to an active Publisher with Write API-Key.vi.

Alternatively you can use API-Key property node as shown below:

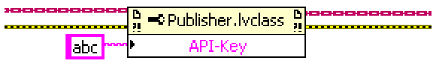


Figure 20 – Changing API-Key to an active Publisher using API-Key property node.

Changing max DT

When subscribers try to connect, the request includes their system time. If subscriber’s system time differs from publisher’s system time more than **maxDt** and **maxDt** is greater than zero, connection is refused. By default, **maxDt** value is zero so subscriber’s system time is not checked. To enable time control, use **maxDt** property node to set the absolute value of maximum distance, in seconds, between local and remote system's time.

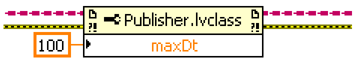


Figure 21 – Changing maxDt to check subscriber's system time.

Publishing and receiving custom data (supported only by LabVIEW applications)

If both side applications use SCCT for LabVIEW, then some advanced features are available. These features rely on LabVIEW variant data type and their data package are not supported on other programming languages. if more than one custom data type is used, custom data types must be associated to a numeric code or string description so that subscribers can properly identify what data has been sent from publisher. In the following example, publisher side, on the left, encode two different data types and send them with **publishData.vi**. Notice they are associated to code 1 and 2. Subscriber application, on the right, receive the packages with **Read customData.vi**, and uses code field to properly process data type 1 and data type 2.

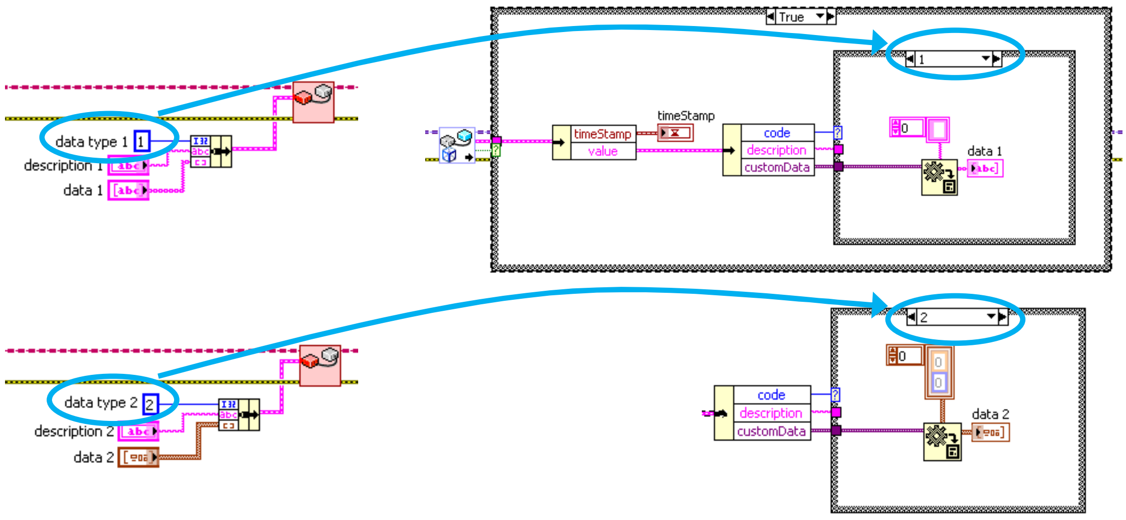


Figure 22 – publisher application and subscriber application use code field to identify the proper data type. Both apps have to know the data type associated to the codes.

Appendix A - Error table

In the following tables are indicated the error codes generated by VIs

Publisher error codes

code	description	explanation
5000	Publisher is not started	A method or property has been called before creating the Publisher
5001	Invalid port	When you start the publisher you have to provide a valid TCP port.
5002	invalid message text	You cannot send a message with an empty text message
5003	Invalid API-Key	API-Key cannot be an empty string
5004	Invalid configuration	Configuration cluster must contain at least 1 analog channel or 1 digital line
5005	Invalid analog data	Analog data must be a 2D array with exactly the number of rows equal to the number of analog channels as defined in configuration cluster.
5006	Invalid digital Data	Digital data must be a 1D array with size equal to the number of digital lines as defined in configuration cluster
5007	Empty data are not published	Empty arrays are not published
5008	internal library is corrupted	Unexpected error during creation of communication tasks

Subscriber error codes

code	description	explanation
6000	Connection is not established	A method or property has been called before establishing the connection or it has been closed
6001	Invalid port	When you start the publisher you have to provide a valid TCP port.
6002	invalid timeout	Timeout must be a value greater than zero
6003	Invalid API-Key	API-Key cannot be an empty string
6004	Connection cannot be created twice	openConnection.vi cannot be used on an active connection. You have to close a connection before re-open
6005	Connection refused from Publisher: wrong API-Key	Publisher refused the connection because API-Key is not correct
6006	Publisher not found	No answer from specified address: possible reasons: publisher is down or not reachable
6007	Publisher Address cannot be empty	Publisher Address cannot be an empty string

Index